

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**System and Method to Specify Extended Configuration
Descriptor Information in USB Devices**

Inventor(s):

Kenneth D. Ray

Firdosh K. Bhesania

John C. Dunn

ATTORNEY'S DOCKET NO. MS1-704US

RELATED APPLICATIONS

This application is related to a prior US Patent Application filed February 4, 2000, titled "Host-Specified USB Device Requests", serial number 09/498,056, which is hereby incorporated by reference.

TECHNICAL FIELD

The following description relates generally to the use of peripheral devices with software applications. More specifically, the following description relates to the use of device specific information and resources with such software applications.

BACKGROUND OF THE INVENTION

The Universal Serial Bus (USB) is a cable bus that supports data exchange between a host computer and a wide range of simultaneously accessible peripheral devices. The attached peripheral devices share USB bandwidth through a host-scheduled, token-based protocol. The bus allows peripherals to be attached, configured, used, and detached while the host and other peripherals are in operation.

The USB is defined by a specification that is approved by a committee of industry representatives. The specification covers all aspects of USB operation, including electrical, mechanical, and communications characteristics. To be called a USB device, a peripheral must conform to this very exacting specification.

USB device information is typically stored in so-called "descriptors" or request codes—data structures formatted as specified by the USB specification. Descriptors are used in a USB system to define "device requests" from a host to a

1 peripheral device. A device request is a data structure that is conveyed in a
2 “control transfer” from the host to the peripheral device. A control transfer
3 contains the following fields:

- 4 • *bmRequestType*—a mask field indicating (a) the direction of data
5 transfer in a subsequent phase of the control transfer; (b) a request
6 type (standard, class, vendor, or reserved); and (c) a recipient
7 (device, interface, endpoint, or other). The primary types of
8 requests specified in the “request type” field are the “standard”
9 and “vendor” types, which will be discussed below.
- 10 • *bRequest*—a request code indicating one of a plurality of different
11 commands to which the device is responsive.
- 12 • *wValue*—a field that varies according to the request specified by
13 *bRequest*.
- 14 • *wIndex*—a field that varies according to request; typically used to
15 pass an index or offset as part of the specified request.
- 16 • *wLength*—number of bytes to transfer if there is a subsequent data
17 stage.

18 All USB devices are supposed to support and respond to “standard”
19 requests—referred to herein as “USB-specific” requests. In USB-specific
20 requests, the request type portion of the *bmRequestType* field contains a
21 predefined value indicative of the “standard” request type.

22 Each different USB-specific request has a pre-assigned USB-specific
23 request code, defined in the USB specification. This is the value used in the
24 *bRequest* field of the device request, to differentiate between different USB-
25 specific requests. For each USB-specific request code, the USB specification sets

1 forth the meanings of *wValue* and *wIndex*, as well as the format of any returned
2 data.

3 USB devices can optionally support “vendor” requests—referred to herein
4 as “device-specific” requests. In a device-specific request, the request type
5 portion of the *bmRequestType* field contains a predefined value to indicate a
6 “vendor” request type. In the case of device-specific requests, the USB
7 specification does not assign request codes, define the meanings of *wValue* and
8 *wIndex*, or define the format of returned data. Rather, each device has nearly
9 complete control over the meaning, functionality, and data format of device-
10 specific requests. Specifically, the device can define its own requests and assign
11 device-specified request codes to them. This allows devices to implement their
12 own device requests for use by host computers, and provides tremendous
13 flexibility for manufacturers of peripherals.

14 The inventors have discovered a need for a similar feature that would
15 benefit various hosts, application programs, host operating systems hardware
16 manufacturers (OEMs), and Independent Hardware Vendors (IHVs). Specifically,
17 designers of application programs and operating systems would value the
18 opportunity to define their own device requests (and the associated responses), and
19 to have such requests supported in a uniform way by compatible peripherals.
20 Moreover, OEMs and IHVs would value the ability to supply additional USB
21 device specific information to the hosts, application programs and host operating
22 systems in response to such device requests. However, the different request types
23 supported in the *bmRequestType* field of a USB device request do not include a
24 “host” type of request.
25

1 As an example of this need for a host type of request, consider that the USB
2 Device Working Group (DWG) has defined a set of standards that describe
3 different types of USB devices. For each of these standards, the DWG has created
4 a respective class specification that specifies static predetermined class and
5 subclass numbers that correspond to the class specification. These specifications
6 are designed such that to be in compliance with a particular class specification, an
7 operating system must include only a single default, or "generic" device driver to
8 control a USB device that is configured according to the particular class
9 specification.

10 When a USB device is installed onto a system, the device indicates that it
11 was created according to a particular DWG class specification by listing
12 predefined class and subclass codes that correspond to the DWG specification.
13 These predefined codes can be used by an operating system to load a single
14 generic device driver to control the device.

15 However, it is often the case that a vendor of a USB device would prefer
16 that a different, or "more specific" device driver be used to control the device,
17 rather than the generic device driver supplied by the operating system. For
18 example, a more specific device driver may be preferred to work around known
19 bugs in the device's software, provide product identification, use value added
20 features, and the like. Such more specific device drivers are typically supplied by
21 OEM/ITHV supplied installation media that are distributed with a device. Such
22 installation media include installation disks (e.g., floppy diskettes) or setup
23 computer program applications.

24 To provide for more specific device driver matches with respect to a USB
25 device, USB device descriptors include markings that describe the manufacturer,

1 product, and revision. An operating system (OS) can use such markings to
2 determine whether a more specific, and therefore a more customized device driver
3 match is available from the installation media. If for some reason a specific match
4 is not found, the concepts of class and subclass codes, as described above,
5 facilitate the loading of a generic device driver to control the device.

6 One of the benefits of using a USB device is the reduced amount of
7 interaction typically required of a user to attach and configure a device. This has
8 been a strong point of USB and has lead to increasing sales of USB devices over
9 the past few years. Thus, being able to load a device driver without having to use
10 additional OEM/IHV supplied installation media to configure the device greatly
11 simplifies end user USB device attachment scenarios.

12 Unfortunately, unless the USB DWG has already allocated certain class and
13 subclass codes for a particular USB device, that particular device cannot have a
14 match based on class and subclass codes to identify a generic device driver to
15 control the device. If the device's corresponding class and subclass codes are not
16 supported by the DWG, an OEM/IHV distributing the device must at least provide
17 installation media with the device to specify a given match that corresponds to a
18 generic driver that is already present on the system. Otherwise, the OEM/IHV
19 must supply a special device driver with the corresponding installation media.

20 In light of the above, a user of a device that specifies non-standard DWG
21 class codes is required to use such installation media to load a device driver to
22 control the device. It would be beneficial both in terms of simplifying matters for
23 OEMs/IHVs and simplifying customer ease of use if an operating system could
24 specify a default device driver that is already present on the system in response to
25

1 installing a device that does not specify a standard USB DWG class and/or
2 subclass code.

3 Ideally, an operating system vendor could develop a new standard USB
4 device class—one not supported by DWG, and distribute a device driver to control
5 devices built to that new standard with the operating system. In this manner, an
6 OEM/IHV could not only distribute devices built to that standard before the USB
7 DWG has adopted a corresponding class and subclass code, but also distribute
8 such devices without needing to supply a special device driver and corresponding
9 installation media. In this way the operating system would natively support the
10 new non-standard USB device by providing a generic device driver to control the
11 new device.

12 To natively support a non-standard USB device, an operating system would
13 have to request specific information from the new device to identify the new
14 device's non-standard class and subclass codes to determine the particular generic
15 device driver to control the new device. However, in response to device specific
16 requests from an operating system, a USB device (as currently defined by the USB
17 specifications) can only return information related to vendor identification,
18 product identification, and revision information. There is no standard request to
19 allow a vendor to return additional information to an operating system to indicate
20 that the device supports non-standard class and/or subclass codes—those not
21 supported by the USB DWG.

22 Thus, it would be beneficial if a USB device could indicate through a
23 predefined USB request that it supports such non-standard class and/or subclass
24 codes. This information could be used by an operating system to install default
25

1 device drivers to control the USB device. However, as discussed above, such a
2 USB request is not defined by the USB specification.

3 In yet another example that illustrates the need for a host specific USB
4 request, consider that the DWG has established a number of well-defined requests
5 to obtain interfaces that correspond to functions in certain composite devices. A
6 composite device includes a number of logical devices (LDs), wherein each LD is
7 a fully functional sub-device of the composite device. There is typically a one-to-
8 one correspondence between a function and an interface in a composite device,
9 with the exception of certain devices such as USB speaker devices, which
10 implement more than one interface for a single function. In such exceptional
11 cases, because the DWG has a predefined request to return the multiple interfaces
12 that correspond to a function, each interface can be returned in response to a single
13 request.

14 Otherwise, when the DWG has not defined such a request, a complex
15 procedure must typically be implemented to determine each of the interfaces that
16 correspond to respective functions in a composite device. For example, to
17 determine if a USB device is a composite device, a host must first request the
18 device to indicate how many functions, or LDs that it supports. In response, a
19 composite device will indicate its ability to support multiple functions by first
20 identifying the number of logical devices (LDs) that the device supports. Next, to
21 obtain specific information corresponding to each supported LD, the host must
22 typically issue separate requests, each request being overloaded with a respective
23 LD identification, to obtain each respective LD descriptor. Thus, a typical
24 composite USB device would not return information corresponding to each
25 interface grouping defined in the composite device in response to a single query

1 unless the single query was already predefined by the DWG to return more than
2 one interface in response to a single request.

3 It would be beneficial to USB composite device makers and to software
4 developers if there was a simple way for a USB device to specify the interfaces
5 that comprise a single function, such that the interfaces could be determined
6 without the needing to implement the complex procedure discussed above. In this
7 manner, newer composite devices not yet supported by the DWG could be treated
8 exactly as the ones currently supported by the DWG.

9 Accordingly, the invention arose out of concerns associated with providing
10 a host-specific device request that solves the problems described above.

11 12 SUMMARY

13 One embodiment of the described system and procedure specifies non-
14 standard compatible IDs, or non-standard class and subclass codes in a USB
15 device in response to a host-specific device request. A non-standard compatible
16 ID is a class or subclass code that is not supported or defined by the USB DWG.
17 The USB device comprises an extended configuration descriptor in firmware of
18 the USB device. The extended configuration descriptor comprises a set of non-
19 standard class codes that can be used by a host operating system to identify one or
20 more device drivers to control the device. Responsive to receiving a host-specific
21 device request from the operating system, the USB device communicates the
22 extended configuration descriptor to a requestor. Thus, the system and procedure
23 of this implementation simplifies end-user device installation scenarios and allows
24 OEMs/IHVs to implement new USB devices without distributing special device
25 drivers and/or corresponding installation media with each single new USB device.

Moreover, the extended configuration descriptor includes a header section and one or more control function sections. The header section indicates the number of control functions for which mappings exist in the extended configuration descriptor. Each control function section indicates information corresponding to a respective function for the USB device. Each control function comprises one or more interfaces. Because the extended configuration descriptor is communicated to a requestor responsive to the receipt of a single host-specific device request, the inventive concepts of this implementation provide a simple way for a new USB device to specify a group of interfaces that comprise a single function.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a block diagram of an exemplary host/peripheral USB system.

Fig. 2 is a flowchart diagram that shows top-level methodological aspects of an exemplary procedure to obtain and use a device-specific request code for host-specific device requests.

Fig. 3 is a flowchart diagram that shows detailed aspects of an exemplary procedure to obtain a device-specific request code.

Fig. 4 is a block diagram showing aspects of an exemplary data structure for an extended configuration descriptor.

DETAILED DESCRIPTION

The following description sets forth a specific embodiment of a system and procedure that incorporates elements recited in the appended claims. The embodiment is described with specificity in order to meet statutory requirements.

1 However, the description itself is not intended to limit the scope of this patent.
2 Rather, the inventors have contemplated that the claimed subject matter might also
3 be embodied in other ways, to include different elements or combinations of
4 elements similar to the ones described in this document, in conjunction with other
5 present or future technologies.

6 7 Exemplary System

8 Fig. 1 shows a system 100 wherein a USB device includes non-standard
9 compatible IDs, or non-standard class and subclass codes. Additionally, the
10 system includes a host application program and host operating system that is able
11 to enumerate non-standard compatible IDs, or non-standard class and subclass
12 code corresponding to USB devices. System 100 is compatible with the Universal
13 Serial Bus (USB) specifications. These specifications are available from USB
14 Implementers Forum, which has current administrative headquarters in Portland,
15 Oregon (current Internet URL: www.usb.org).

16 System 100 includes a host computer 102 and a USB peripheral device 114.
17 The respective functionality of the computer and peripheral device is embodied in
18 many cases by computer-executable instructions, such as program modules, that
19 are executed by respective processors. Generally, program modules include
20 routines, programs, objects, components, data structures, etc. that perform
21 particular tasks or implement particular abstract data types.

22 Computer 102 is a conventional desktop PC or other type of computer.
23 Computer 102 has one or more processors 104 and one or more forms of
24 computer-readable memory media 106 such as electronic memory, magnetic
25 storage media, optical storage media, or some other type of data storage.

1 Programs are stored in memory 106 from where they are executed by processor
2 104. In this example, such programs include an operating system 108 such as the
3 Microsoft "Windows"® family of operating systems. The operating system
4 provides various system services to one or more application programs 110 running
5 on the computer.

6 The computer also has a USB communications driver and port 112. The
7 USB port is supported by operating system 108. To communicate with a USB
8 device, an application program 110 makes high-level calls to system services
9 provided by the operating system. The system services take care of lower level
10 communications details, and return requested information to the application
11 program.

12 Peripheral device 114 is one of any number of different types of USB
13 devices such as a data storage device, a digital camera, a scanner, a joystick, game
14 pad, steering unit, mouse, stylus, digital speaker, microphone, display device, and
15 the like. Peripheral device 114 has one or more processors 116 and one or more
16 forms of computer-readable memory media 118, including at least some form of
17 non-volatile memory media 120.

18 The peripheral device 114 has a USB port 126 and communicates with
19 computer 102 via a USB communications medium 128. The peripheral device
20 also has operating logic 124, which is executed by processor 116 to detect control
21 actuation and for communicating with computer 102 across communication path
22 128.

23 The peripheral device 114 responds to requests from the host computer 102
24 across the communication path 128. These requests are made using control
25 transfers where setup packets (not shown) are exchanged. The USB device returns

1 descriptors in response to exchanging such setup packets. Although the USB
2 Specification defines a number of different standard class and vendor specific
3 descriptors, an extended configuration descriptor 122 is not defined in the USB
4 specification. The extended configuration descriptor allows OEMs/IHVs to store
5 non-standard class and subclass codes—those that are not yet defined by the USB
6 DWG, into non-volatile memory 120 of the device. Moreover, the extended
7 configuration descriptor provides a simple way for a composite USB device to
8 specify a group of interfaces that comprise a single function.

9 Extended configuration descriptor 122 provides non-standard class codes
10 and subclass codes on a per function basis. (Device vendors can build multiple
11 functions into a device). A function is an independently controlled aspect of a
12 device. A function can be considered to be a group of interfaces conglomerated
13 together that serve a similar purpose. Class codes are also referred to herein as
14 “compatible ids”. Subclass codes are also referred to herein as “sub-compatible
15 ids”.

16 Peripheral device 114 supports host-specific device requests to obtain
17 information that is not specified in the USB specification. In this implementation,
18 prior to using the peripheral device, the computer sends a host-specific request to
19 the peripheral device requesting class codes and/or subclass codes that correspond
20 to the peripheral device. In response to the request, the peripheral device provides
21 extended configuration descriptor 122 to the computer. Within the extended
22 configuration descriptor, the computer locates data corresponding to the peripheral
23 device 114, and extracts non-standard class and/or subclass codes, as well as one
24 or more interfaces that correspond to one or more functions. The operating system
25

1 then loads device drivers based on the extracted extended configuration descriptor
2 to control and interface with the peripheral device.

3 4 Exemplary Procedure to Obtain a Device Specific Request Code

5 Fig. 2 shows top-level methodological aspects of an exemplary procedure
6 to obtain a device-specific request code for a computer to use when making a host-
7 specific device request. Generally, a new non-standard USB-specific device
8 request is defined for use with various USB peripherals. This request is referred to
9 herein as a host-specific device request. Because of the described methodology,
10 the host-specific device request can be defined by the manufacturer of an
11 operating system or application program, and can then be made available to
12 peripheral vendors so that they can support and respond to the newly defined
13 request. As an example, an OS manufacturer might define a new descriptor
14 allowing peripherals to return device specific data and resources to an operating
15 system in a data format that is determined by the operating system. This would
16 allow the operating system to use a single device request to obtain this information
17 from various different peripherals (assuming those peripherals are configured to
18 support the new device request).

19 In an initialization phase 200, the host sends a request to the peripheral in
20 the form of a USB-specified device request. The request is for a device-specific
21 request code—of the device's choosing—that will be subsequently used as the
22 request code for the host-specific device request.

23 Once this request code is obtained, it is used in a subsequent phase 202 to
24 initiate the host-specified device request to obtain the extended configuration
25 descriptor 122 (see Fig. 1). Specifically, the host specifies the request code as the

1 *bRequest* value in a control transfer (see the “Background” section for details
2 regarding a control transfer). The actual protocol of this device request (meanings
3 of *bIndex*, *bValue*, etc.) is as specified in the definition of the host-specific device
4 request. Phase 202 is repeated as desired during subsequent operation, without
5 repetition of initialization phase 100.

6 Fig. 3 shows more details regarding the initialization phase 200. The host
7 performs an action 300 of sending a GET_DESCRIPTOR device request to the
8 peripheral device. The GET_DESCRIPTOR device request is a standard, USB-
9 specific request, identified in a control transfer by the GET_DESCRIPTOR
10 request code (*bRequest* = GET_DESCRIPTOR). The fields of the control transfer
11 (discussed above in the background section) have values as follows:

- 12 • *bmRequestType* = 10000000 (binary), indicating a “device-to-
13 host” transfer, a “standard” or “USB-specific” type request, and a
14 device recipient.
- 15 • *bRequest* = GET_DESCRIPTOR. This is a constant (six) defined
16 by the USB specification
- 17 • *wValue* = 03EE (hex). The high byte (03) indicates that the
18 request is for a “string” descriptor, and the low byte is an index
19 value that is predefined as a constant in the definition of the host-
20 specified device request. In this example, it has been defined as
21 EE (hex), but could be predefined as any other value.
- 22 • *wIndex* = 0.
- 23 • *wLength* = 12 (hex). This is the length of a host-specific request
24 descriptor that will be returned in response to this request. In the
25 described example, the length is 12 (hex).

- *data*—returned host-specific request descriptor.

A compatible USB device is configured to respond to a request such as this (where *wValue* = 03EE (hex)) by returning a host-specific request descriptor such as the extended configuration descriptor 122 of Fig. 1. This descriptor is not defined by the USB standard, but has fields as defined in the following discussion. The host-specific request descriptor designates a device-specific request code that will work on *this device* to initiate the host-specific request code. In other words, the manufacturer of the device can select any device-specific request code, and associate it with an implementation of the host-specific device request.

More specifically, the device receives the GET_DESCRIPTOR device request (block 302) and performs a decision 304 regarding whether the index value (the second byte of *wValue*) corresponds to the predetermined value (EE (hex)). This predetermined value is a value that is chosen to be used specifically for this purpose.

If the index value does not correspond to the predetermined value, the device responds at 305 in an appropriate way, usually by returning some other descriptor that corresponds to the index value. If the index value does correspond to the predetermined value, an action 306 is performed of returning the host-specific request descriptor to the host.

The host-specific request descriptor includes, for example, the following fields:

- *bLength*—the length of the descriptor (12 (hex) in this example).
- *bDescriptorType*—the type of descriptor (string type in this example).

- *qwSignature*—a signature confirming that this descriptor is indeed a descriptor of the type requested. The signature optionally incorporates a version number. For example, in the described example MSFT100 indicates that this descriptor is for an “MSFT” host-specific device request, version “100” or 1.00.
- *bVendorCode*—the device-specific request code that is to be associated with the host-specified device request.
- *bPad*—a pad field of one byte.

At 308, the host receives the host-specific request descriptor and then performs an action 310 of checking or verifying the signature and version number found in the *qwSignature* field. The correct signature confirms that the device is configured to support host-specific request codes. If either the signature or version number is incorrect, at 311 the host assumes that the device does not support host-specific request codes, and no further attempts are made to use this feature.

The signature field of the host-specific request descriptor block is what provides backward compatibility. A non-compatible device (one that doesn’t support host-specific request codes) might use the predetermined *wValue* 03EE (hex) to store some other string descriptor, which will be returned to the host without any indication of problems. However, this will become apparent to the host after it examines the data in the location where the signature is supposed to be. If the signature is not found, the host knows that the returned descriptor is not of the type requested, and will assume that the device does not support host-specific request codes.

1 If the signature and version are confirmed in block 310, at block 312 the
2 host reads the device-specific request code from the *bVendorCode* field, and uses
3 it in the future as a host-specific request code, to initiate the host-specific device
4 request. When using the device, the host sends the host-specific device request by
5 specifying the obtained device-specific request code as part of a control transfer.
6 The device responds by performing one or more predefined actions or functions
7 that correspond to the host-specific device request, in accordance with the
8 specification of the host-specific device request.

9 The host-specific device request itself is in the format of a normal USB
10 control transfer, including the fields discussed in the "Background" section above.
11 The *bRequest* field is set to the value of the *bVendorCode* field of the host-specific
12 request descriptor, which was earlier obtained from the peripheral device. The
13 *bmRequestType* field is set to 11000001 (binary), indicating a device-to-host data
14 transfer, a "vendor" or device-specific request type, and a device recipient.

15 The *wValue* and *wIndex* fields are used as defined by the definition of the
16 host-specific device request. The *wLength* field indicates the number of bytes to
17 transfer if there is a subsequent data transfer phase in the device request.

18 In a presently preferred implementation of this system, the host-specific
19 device request is used to request one of a plurality of available host-defined string
20 descriptors from the device. The *wIndex* field of the host-specific device request
21 indicates which of the plurality of strings are to be returned. The device returns
22 the descriptor referred to by *wIndex*.

23 The techniques described above allow an operating system designer to
24 specify informational descriptors that devices can implement to provide additional
25 data about themselves—data that is not directly addressed by the USB

1 specification. For example, the techniques described above allow an operating
2 system to specify the extended configuration descriptor 122 of Fig. 1. The
3 operating system uses the extended configuration descriptor to enumerate all USB
4 devices that support non-standard USB DWG class codes and/or subclass codes to
5 determine one or more device drivers to control the devices. The extended
6 configuration descriptor and procedures to use the extended configuration
7 descriptor to enumerate non-standard compatible ids are described in greater detail
8 below in reference to Fig. 4 and Tables 1-2.

9 10 Exemplary Data Structures

11 Fig. 4 shows elements of an exemplary data structure for the extended
12 configuration descriptor 122, which is an example of the host-specific request
13 descriptor described above. The extended configuration descriptor 122 includes a
14 header section 400 and a function section 402. The header section includes a
15 number of elements 404 that describe the function section. The function section
16 includes one or more control functions 406. Each instance of a control function
17 406 encapsulates information corresponding to a single function for the peripheral
18 device 114.

19 Header section 400 stores information about the remaining portions of the
20 extended configuration descriptor 122. Header section 400 includes the following
21 fields:

- 22 • *dwLength*—the total length of the extended configuration descriptor;
- 23 • *bcdVersion*—the version number of the extended configuration
24 descriptor;
- 25 • *wIndex*—identifies the extended configuration descriptor; and

- *wCount*—the total number of control function sections 406 in the function section 402.

Using this information, the operating system or application program can parse the following function section.

Function section 402 is of variable size because it includes one or more control function sections 406. Each control function section includes information that corresponds to a single function for the peripheral device. The function section includes the following fields:

- *dwSize*—the total length of this control function section;
- *qwCompatibleID*—the compatible ID, or class code for this function;
- *qwSubCompatibleID*— the sub-compatible ID, or subclass code for this function;
- *bTotalInterfaces*—the number of interfaces grouped together to generate this function;
- *bInterfaceNumber*—the interface number;

The *qwCompatibleID* and *qwSubCompatibleID* values are used to override any USB DWG standard compatible ids and/or sub-compatible ids that may be defined for the device. This means that if USB DWG standard class and/or subclass codes are defined for the USB device, that these standard codes will be ignored in view of the extended configuration descriptor provided compatible and/or sub-compatible ids. The *qwCompatibleID* and *qwSubCompatibleID* values should not be confused with the USB DWG defined class and subclass codes—regardless of whether they are used in the extended configuration descriptor.

The *bTotalInterfaces* field is a value that represents the total number of interfaces that have been grouped together for this particular function. Each individual interface number (*bInterfaceNumber*) forming the function is represented below it.

Example Extended Configuration Descriptor Use

For an operating system to use an extended configuration descriptor stored on a device, the device must first provide information to the operating system that indicates that the device supports host-specific device requests. In this implementation, this is accomplished by storing the string descriptor shown in TABLE 1 at index 0xEE on the USB device. The string descriptor defines an operating system descriptor that is returned to the operating system in response to a USB device request as discussed above. In this implementation, the string descriptor index is 0xEE, however it could be stored at some other location in the USB device.

TABLE 1
Example of an Operating System String Descriptor

```
const byte os_descriptor[]={  
    0x12,                //length  
    0x03,                // string descriptor type  
    _M_,_S_,_F_,_T_,_1_,_0_,_0_, //data  
    0x01,                //index  
    0x00                //reserved  
};
```

Next, the extended configuration descriptor is stored in a format specified by the operating system on the firmware of the USB device. The extended

configuration descriptor is shown below in TABLE 2. In this implementation, the USB device is a device that is compatible with a Remote Network Device Interface Specification (RNDIS) that is not supported by the USB DWG. (This means that the USB DWG has not supplied any compatible ids for RNDIS devices). In this implementation, the device needs to indicate to the operating system that it is compatible with the RNDIS. To accomplish this, the device implements the following extended configuration descriptor.

TABLE 2
Example of an Extended Configuration Descriptor for a RNDIS
Compatible USB Device

```
const byte mem_stick_descriptor[]={ //header section
    0x00000044, // dwLength (DWORD)
    0x01000, // bcdVersion
    0x0004, // wIndex
    0x0001, // wCount

    // function section
    0x00000016, // wSize
    "RNDIS", // qwCompatibleID
    "", // qwSubCompatibleID
    2, // bTotalInterfaces
    1, // First Interface in function 1
    3 // Second interface in function = 3
};
```

When the RNDIS compatible device is attached to the USB port on the computer, the operating system retrieves the extended configuration descriptor from the device by using the following API call:

```
GET_DESCRIPTOR(
    bmRequestType = 1100 0001B, // device-to-host data transfer
    bRequest = 0x01, // this request
```

wValue = 0x0000,
wIndex = 0x0004);

// string descriptor indication
// this index

In response to receiving this request, the USB device returns the extended configuration descriptor to the operating system. Upon receiving the extended configuration descriptor, the operating system determines the non-standard compatible ids based on the extended configuration descriptor. The operating system uses the non-standard compatible ids to load generic (or default) device drivers to control the USB device. This is beneficial because OEMs/IHV's who develop USB devices that are not fully supported by USB DWG standard class and subclass device drivers can specify non-standard class and subclass device drivers to control the USB devices.

Moreover, unless the DWG has a predefined request that returns one or more multiple interfaces that correspond to a function in a composite USB device, a complex procedure must typically be implemented in traditional systems and procedures to determine the interfaces that correspond to the functions. However, the extended configuration descriptor of this implementation includes each interface to function mapping implemented by a composite device. Thus, in contrast to such traditional systems and procedures, this implementation provides a simple way for a new composite USB device to specify a group of interfaces because the extended configuration descriptor is communicated to a requestor in response to receipt of a single host-specific device request.

Although the invention has been described in language specific to structural features and/or methodological steps, it is to be understood that the invention defined in the appended claims is not necessarily limited to the specific features or steps described. Rather, the specific features and steps are disclosed as preferred forms of implementing the claimed invention.